MakerGame LRM

Cindy Wang (xw2368),
Steven Shao (ys2833),
Yuncheng Jiang (yj2433)

Introduction

Inspired by Game Maker, MakerGame is language specialized for 2D video games with a C-like syntax. Most of the primitives in C are present in MakerGame, but the central construct in MakerGame is the object type, similar to a struct in C, but with added game event functions to specify the behaviors of game objects through their lifetimes. Game programming patterns like the event loop are automatically taken care of.

Lexical Conventions

Comments

Block-comments are introduced with `/*` and terminated with `*/`. They cannot be nested.
Line-comments start with a `//` and continue to the end of the line.

Identifiers

Identifiers start with an alphabetic character, followed by any number of alphanumerics or underscores. The following keywords cannot be used as identifiers:

`create`	`destroy`	`draw`	`step`	`exists`
`if`	`else`	`for`	`while`	`foreach`
`break`	`continue`	`return`	`include`	
`int`	`bool`	`char`	`float`	
`void`	`sprite`	`sound`	`true`	`false`

Literals

1. **Integer** literals are nonempty sequences of digits from 0 to 9.
2. **Floating point** literals consist of a sequence of digits, then a decimal point, then optionally another sequence of digits.

3. **Boolean** literals are either `true` or `false`.
4. **Character** literals consist of an ASCII character (minus single and double quotes) surrounded by single quotes.
5. **String** literals are any sequence of ASCII characters (minus single and double quotes) surrounded by double quotes.

Operators

In order of decreasing precedence,

operator	**associativity**	**description**
`() []`	left-to-right	function call, array subscripting
`.`	left-to-right	object member access
`! - (unary)`	right-to-left	logical and numerical negation
`^`	left-to-right	exponentiation
`* / %`	left-to-right	multiplication, division, modulo
`+ - (comp)`	left-to-right	addition, subtraction
`== <= >= < >`	left-to-right	boolean relations
`&& ||`	left-to-right	Logical AND, logical OR
`=`	right-to-left	assignment

Misc. punctuation

Some punctuation also act as separators or parts of more complex structures.

1. **Parentheses** group expressions in order to force precedence order.
2. **Commas** separate function arguments and elements in an array definition.
3. **Semicolons** terminate statements.
4. **Curly Braces** enclose code blocks (statements and definitions); object definitions; and array definitions.
5. **Whitespace** separates tokens.

Datatypes

Primitives

Following is a list of data types and sample declarations (`type ident = literal`).

1. **int**: 32-bit signed integer value
   `int x = 5;`
2. **float**: 32-bit floating point value
   `float y = 3.14;`
3. **bool**: one of ‘true’ or ‘false’
   `bool b = true;`
4. **char**: an 8-bit ASCII character
   `char c = '3';`
5. **sprite**: an opaque handle to an image that can be drawn to the screen.
   `sprite spr = load_sprite("player.png");`
6. **sound**: an opaque handle to a sound file that can be played.
   `sound snd = load_sound("level.ogg");`

Collections

Fixed-length arrays

Arrays are a contiguous sequence of objects of a particular type. They can be instantiated with

```
        typename[size] identifier    or    typename[] identifier
```

where the latter is allowed if the size can be inferred from subsequent assignment. Examples:

```
int[5] arr;
int[2][3] mat;
int[] seq = {1, 2, 3};
char[] hello = "hello";
char[] bye = {'b', 'y', 'e'};
```

Game objects and rooms

Game objects are a collection of primitive types and arrays, combined with four event handlers with fixed names that define the object's behaviour in the game. Their generic definition is as follows:

```
objectname {
      // variable declarations
      type1 name1; type2 name2; // etc.
      // all definitions are optional, with defaults empty

      // create can have parameters; not the others.
      CREATE(type1 arg1, type2 arg2, ...) { /* … */ }
      DESTROY { /* … */ }
      STEP { /* … */ }
      DRAW { /* … */ }
}
```

These objects can be instantiated, and every frame, the body of the STEP handler of every instantiated object is run, followed by the body of every DRAW handler for rendering. An object's CREATE and DESTROY event handlers are called when the object is instantiated and removed from the game, respectively.

Bodies of event handlers have access to both member variables, locally declared variables, and global variables. One can also refer to the calling object itself with `this`.

Example of object creation and destruction:

```
Player p = create(Player);
destroy(p);
```

An object name can be annotated with the string "`@ROOM`" to indicate that it is of a special room type that cannot be `creat`ed, but rather `start`ed (defined in Built-in Functions section).

Functions

User-defined Functions

One can define a function in the global namespace with the following syntax:

```
returntype functionname(type1 arg1, type2 arg2, …) { /* fn body */ }
```

The function body can refer to locals, arguments, and globals. If a particular variable refers to a game object type, one can access the member variables of that game object with the `.` operator.

Built-In Functions

MakerGame includes the following built-in functions by default:

1. Window (`set_window_size(w, h), set_window_fps(fps), exit()`)
2. Resources
    a. `load_sprite(filename), load_sound(filename)`
    b. `draw_sprite(x, y, sprite)` - render an image to a location on the screen
    c. `draw_text(x, y, text, r, g, b)` - render text to the screen in a particular color
    d. `play_sound(sound, repeat)` - play the sound, possibly repeating
3. Objects
    a. `create(Object, arg1, arg2, …)` - create and get a handle to a new instance of Object, constructed with specific arguments.
    b. `start(RoomObject, arg1, arg2, …)` - remove all objects in the game, and instantiate the given room.
    c. `destroy(object)` - remove the object referenced by this handle from the game
    d. `exists(object)` - check if the handle references an object still in the game

In addition, MakerGame has libraries for more specialized functions:

1. Input (`key_down(key), mouse_down(), mouse_x(), mouse_y()`)
2. Printing (`print(char[]), print(int), print(bool), print(object)`)
3. Math (`sin, sqrt, etc.`)
4. Time (`get_day(), get_seconds()`)
5. Geometry and collision - distances, hitboxes, etc., and maybe type to express hitboxes.

Statements and control flow

Function and object event-handler bodies are code-blocks, which contain a list of statements and declarations. Like in C, statements are either expressions, `return` statements, brace-enclosed code-blocks, or control flow structures. Following is a sample of these flows:

Conditionals

As in C, a conditional statement executes the branch corresponding to the first boolean predicate that's satisfied.

```
if (bool1) {
      statements
} else if (bool2) {
      statements
} else {
      statements
}
```

Loops

As in C, a while loop repeatedly executes its body as long as the boolean expression is true.

```
while(bool) { statements }
```

A for loop is functionally equivalent to a while loop with a statement (*stmt1*) before its body and a statement at the end of each iteration of its body (*stmt2*).

```
for (stmt1; pred; stmt2) { statements }
```

A foreach loop iterates through every object of a particular game object type (*objtype*).

```
foreach(objtype identifier) { statements }
```

Program Structure

A program consists of a sequence of declarations of the following types:

1. Global variable declarations
2. Global function declarations
3. Game object declarations

Global functions and variables are in scope by the end of its declaration. Game object type names and member variables are in scope at every point in the file (so that functions can refer to game objects defined later in a file). The built-in functions are globally scoped, and additional functions, objects, and variables from other libraries and other included files are in scope from their point of inclusion.

Each game must have an object called "Game" tagged as a room (`Game@Room`). The resultant program's entry point simply starts the Game room.

Sample Program

```
// Initializing resources & global variables
Sprite playerSprite = load_sprite(“player.png”);
Sprite eggSprite = load_sprite(“egg.png”);
Sound boinkSound = load_sound(“boink.ogg”);

// Definitions of global functions
// notice how these functions can operate on game object variables
bool hit_ground(Egg e) { return (e.y > 600); }
bool egg_touching_player(Egg e, Player p) {
      return (e.x < p.x + 50 && e.x > p.x - 50
            && e.y < p.y + 10 && e.y > p.y - 10);
}

// Definitions of object types
Egg {
      int x; int y;
      // sound effect upon spawning
      CREATE() { play_sound(boinkSound); }
      // fall 5px every frame, and game over upon hitting the ground
      STEP {
            y += 5;
            if (hit_ground(this)) start(GameOver);
      }
      // render the location of the egg every frame
      DRAW { draw_sprite(x, y, eggSprite); }
}

Player {
      int x; int y;

      CREATE(int X, int Y) { x = X; y = Y; }
      STEP {
            // respond to keyboard input
            if (key_pressed(left)) x -= 5;
            if (key_pressed(right)) x += 5;

            // continuously check if we caught an egg
            foreach (Egg egg) {
                  if (check_touching(obj, this)) {
                        // “catch” the egg and increase score
```

```
                    destroy(egg);
                    score += 5;
                    print(“SCORE: “ + score);
                }
            }
        }

        // every frame, rerender the player at the right place
        DRAW { draw_sprite(x, y, playerSprite); }
}

Spawner {
        int timer = 50;
        int spawn_positions[4] = {100, 200, 300, 400};
        STEP {
            // each frame, decrement timer.
            // when it reaches 0, create an egg somewhere.
            --timer;
            if (timer == 0) {
                timer = 50;
                Egg egg = create(Egg);
                Egg.x = spawn_positions[random(4)];
                Egg.y = 100;
            }
        }
}

Game@Room {
        CREATE() {
            // at start of game:
            //  - put objects into scene/room
            //  - set window parameters
            set_window_size(600, 600);
            Player p = create(Player, 300, 500);
            create(Spawner);
        }
}

GameOver@Room {
        DRAW {
            draw_text(“game over”, 5, 5, 255, 255, 255); // white text
        }
}
```